METHOD FOR ASSEMBLING SUB-POOLS OF TEST QUESTIONS

BACKGROUND OF THE INVENTION

FIELD OF THE INVENTION

[0001] The present invention relates to automated test assembly.

DESCRIPTION OF THE RELATED ART

[0002] The following describes common terms used in test development. "Item" is a term designating a question that can be placed on a test. An item has Item Response Theory ("IRT") parameters (discrimination, difficulty, guessing), a cognitive skill subtype and an associated passage. A passage (or stimulus) is used to introduce a group of items. A passage has the following properties: topic, social group orientation, number of words, type, and related items. A section is a collection of items administered together without break. Every section must satisfy stated constraints that can vary from section to section. A test is a combination of sections satisfying the stated test constraints. In some cases, passages are used. In these cases, each item has one related passage and each passage has one or more related items. As referred to herein, a "sub-pool" can be a test or section or any subset thereof.

[0003] Thus, a test is composed of items (test questions) and passages (stimulus material to which the items refer). A test is scored based on an examinee's responses to the items. If there is a one-to-one correspondence between an item and its passage (or if there is no passage), the item is called discrete. If more than one item is associated with a passage, the item is called set based. A database of items, passages and their associated characteristics is called an item bank or item pool. Usually, the type of a section corresponds to the type of its items. If two tests (or sections) have one or more passages (or items) in common, they are called overlapping (also referred to herein as intersecting); otherwise, they are non-overlapping (also referred to herein as disjoint).

[0004] The last two decades has seen a wide spread usage of automated test assembly at testing agencies. Most practical test assembly problems are NP-Complete. Thus, no polynomial algorithm exists for their solution and a search procedure must be used for large problems. This does not mean that the assembly of a single linear test is difficult. Most test assembly situations do not require the optimization of an objective function.

Test specifications are defined and any combination of items meeting the specifications yields an acceptable test. A typical item pool would give rise to a large number of ways to combine items to make a test. Heuristics methods for test assembly are described in the prior art. In certain prior art methods, a combination of network flow and Lagrangian relaxation for test assembly is utilized. In other prior art methods, the use of a more general mixed integer programming (MIP) code was proposed. The MIP approach is now considered a common technique to assemble tests, although it does not support non-linear constraints.

[0005] Adaptive stochastic search, including simulated annealing, genetic algorithms, tabu search, and Monte Carlo methods (random search, Markov chain simulation), are being successfully used for various practical global optimization problems. This success is due to easy implementation and adaptation to the complex problems. The present invention presents a new test assembler exploiting stochastic methods and supporting linear and non-linear constraints.

[0006] One issue relating to test assembly involves the problem of identifying multiple non-overlapping tests. The problem of finding the maximum number of non-overlapping tests is referred to herein as the *extraction problem*. Taking into account the cost of development and maintenance of each item in a pool, the solution of the extraction problem has great value for a testing agency. The conventional approach is to assemble tests sequentially while removing from the pool any previously used items and passages. This technique can not guarantee the optimal solution to the extraction problem because removal of items can block further assembly of non-overlapping tests. The present invention presents a new approach that does not have this disadvantage.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] Fig. 1 is a flow chart illustrating a method of assembling a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0008] Fig. 2 is a schematic illustrating the search region for a test in accordance with a preferred embodiment of the present invention.

[0009] Fig. 3 is a schematic illustrating reduction of the search region for a test into several search regions for sections in accordance with a preferred embodiment of the present invention.

[0010] Fig. 4 is a schematic illustrating reduction of the search region for a section into several search regions for groups of items in accordance with a preferred embodiment of the present invention.

[0011] Fig. 5A is a schematic illustrating a hierarchical representation of a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0012] Fig. 5B is a flow chart illustrating a method of assembling a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0013] Fig. 5C is a flow chart illustrating a method of assembling a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0014] Fig. 6 is a flow chart illustrating a method of assembling a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0015] Fig. 7 is a schematic illustrating shrinking the search region by way of a greedy algorithm in accordance with a preferred embodiment of the present invention.

[0016] Fig. 8 is a flow chart illustrating a method of assembling a sub-pool of questions in accordance with a preferred embodiment of the present invention.

[0017] Fig. 9 is a schematic illustrating overlapping and non-overlapping sections in accordance with a preferred embodiment of the present invention.

[0018] Figs. 10A and 10B are flow charts illustrating preferred embodiments of methods of assembling multiple non-overlapping sub-pools.

SUMMARY OF THE INVENTION

[0019] The present invention is directed to methods for assembling a sub-pool of questions, from a pool of questions, in connection with creation of a standardized test, wherein the sub-pool satisfies one or more constraints. In one embodiment, a candidate sub-pool is formed by randomly selecting a plurality of questions from the pool. The candidate sub-pool is tested against the constraints. If the constraints are satisfied, the candidate sub-pool is stored as the sub-pool.

[0020] In another embodiment, a hierarchical representation of the sub-pool is created. The hierarchical representation includes a root node and at least one other node. At least one of the other nodes includes a terminal node. The root node is associated with one or more root constraints and each of the other nodes is associated with one or more additional constraints. A candidate question set is formed, which comprises randomly

selecting a plurality of questions from the pool. The candidate question set is tested against the additional constraints associated with the terminal node. If the additional constraints are satisfied, the candidate question set, at least, is tested against the root constraints.

[0021] In a further embodiment, a hierarchical representation of the sub-pool is created. The hierarchical representation comprises a root node and at least two other nodes. At least two of the other nodes each comprise a terminal node. The root node is associated with one or more root constraints and each of the other nodes is associated with one or more additional constraints. A first candidate question set is formed, which comprises randomly selecting a plurality of questions from the pool. The first candidate question set is tested against the additional constraints associated with a first of the terminal nodes. A second candidate question set is formed, which comprises randomly selecting a plurality of questions from the pool. The second candidate question set is tested against the additional constraints associated with a second of the terminal nodes. If the first set of additional constraints and the second set of additional constraints are satisfied, the first candidate question set and the second candidate question set are concatenated to form a combined question set. The combined set, at least, is tested against the root constraints.

[0022] In a still further embodiment, a sequence of ranges is formed, wherein each range in the sequence imposes a constraint on a scalar property of the sub-pool. A vector comprising a plurality of elements is randomly formed, wherein each of the elements in the vector belongs to at least one range from the sequence. A plurality of questions is randomly selected from the pool to form the sub-pool such that each of the scalar properties of the sub-pool is equal to at least one of the elements of the vector.

[0023] In a further embodiment, a candidate sub-pool is formed by randomly selecting a plurality of questions from the pool. It is determined whether the candidate sub-pool satisfies the constraints. If the constraints are not satisfied, the questions of the candidate sub-pool are removed from the pool of questions and the process repeats. If the constraints are satisfied, the candidate sub-pool is stored as the sub-pool.

[0024] The present invention is also directed to a method of assembling a plurality of mutually disjoint sub-pools from a pool of questions in connection with creation of a standardized test, wherein each sub-pool comprises a plurality of questions and satisfies

one or more constraints. A first collection of intersecting sub-pools is assembled. A second collection of sub-pools is extracted from the first collection of sub-pools. The second collection of sub-pools comprises the plurality of mutually disjoint sub-pools. The second collection of sub-pools is stored.

DETAILED DESCRIPTION

[0025] The present invention relates to assembling tests based on random and tabu search techniques. The invention exploits the "divide and conquer" technique and the properties of test development constraints. In addition, the present invention relates to assembling multiple, non-overlapping sections and/or tests from a pool of items.

[0026] Assembling a test, or multiple tests, from a pool of items presents a number of issues. First, with regard to assembling a single test from a pool of items and passages, the test must satisfy the stated constraints. For a standardized linear test, the assembled test is often required to be "parallel" to all previously assembled tests of this type. Two tests are parallel if they have the same score distribution, reliability and validity. Second, for a given pool of items and passages, the maximum number of non-overlapping tests that can be assembled must be estimated. In addition, from the given pool, the maximum number of non-overlapping tests must be extracted. Solving the extraction problem provides a solution for the estimation problem. The following presents a method for addressing the single test assembly problem and building strong lower bound of the extraction problem.

[0027] Test Assembly Constraints

[0028] The following provides exemplary constraints that may be used in connection with test development. Other constraints may be used in connection with the methods of the present invention. These constraints are designed based on commonly used MIP model, but have been made more generic and modified to facilitate description of the random search test assembly of the present invention. The notations used in connection with the constraints are as follows:

I - used to denote an item

P - used to denote a passage

S - used to denote a section, sequence of items

T - used to denote a test, sequence of sections

$\mathfrak{I}$ - sequence of all available items (item pool)

$\mathfrak{R}$ - sequence of all available passages (passage pool)

[0029] For a sequence R (items, passages, sections, test), the following generic functions are used:

1. $Items(R)$ returns a sequence of items in R.
2. $Passages(R)$ returns a sequence of passages in R, for example, if R is a section then the sequence of passages related to items of R is $Passages(R)$.
3. $\|R\|$ returns the number of elements in R.
4. $Score(R)$ returns the expected score of R.
5. $SumWords(R)$ returns sum of number of words of elements of R.
6. $Position(r_i)$ returns allowed position of $r_i \in R$ in sequence R, where i is actual position of r_i. For example if $R = \{P_1, P_2, P_3\}$ is sequence of passages, then for passage P_2 its actual position is 2, but its allowed position can be any integer number from $[1,2]$, see below constraint (9).
7. $Enemies(r)$ returns sequence of elements that can not be used together with r (enemy sequence of r) in R.
8. $TFilter(R,t)$ returns subsequence of R where each element has type t.
9. $STFilter(R,st)$ returns subsequence of R where each element has subtype st.
10. $Filter(R,G)$ returns subsequence of R where each element is related to some element from G, for example, if R is sequence of items and G is sequence of passages, $Filter(R,G)$ is sequence of items from R related to passages from G.
11. $Random(R)$ returns a randomly selected element from R (uniform distribution is used).

[0030] Assuming that $R = \{r_1, r_2, ..., r_M\}$ and each $r_j \in R$ is a range $r_j = \lfloor r_j^L, r_j^U \rfloor$, the following function is used:

$Enumeration(R,s)$ generates a vector L of sequences $L_i = \{i_1, i_2, ..., i_M\}$ such that:

1. $\sum_{j=1}^{M} i_j = s$

2. $\forall i_j \in L_i \rightarrow i_j \in r_j$

[0031] In a test assembly application, all elements are integer and the resulting vector L has reasonable size. This function can be more general if s is a range; then, the constraints to be described can be more flexible. For the sake of clarity, this generalization is not considered here, but will be apparent to those skilled in the art.

[0032] Test Constraints

[0033] The constraints are described below in a form suitable for the formulation of the random search method. The method does not check (i.e., calculate) them all explicitly.

Every test must have a specified number of sections.

$$T = \{S_1, S_2, \ldots, S_M\}, \tag{1}$$

where M is the allowed number of sections in test T and S_i is a section.

Every section has limits on the number of items per section, and there is a limit on the number of items per test.

$$\{\|S_1\|, \|S_2\|, \ldots, \|S_M\|\} \in Enumeration(\{RI(S_1), RI(S_2), \ldots, RI(S_M)\}, N), \tag{2}$$

where N is the allowed number of items in test T and $RI(S_i)$ is the allowed range on the number of items in section S_i.

An item can be used at most once on a test.

$$\forall I_j, I_k \in Items(T) \rightarrow I_j \neq I_k \tag{3}$$

The expected test score must be close to the expected score of all previously administered tests of this type.

$$Score(T) \in RS, \tag{4}$$

where RS is the allowed range of expected test score.

With regard to a section $S \in T$, the following provides exemplary section constraints:

Each section has an allowed number of passages and the number of passages of each type must be within a specific range.

$$
\begin{aligned}
&S_P = Passages(S)\\
&\{\|TFilter(S_P,t_1)\|,\|TFilter(S_P,t_2)\|,\ldots,\|TFilter(S_P,t_K)\|\}\in C\\
&C = Enumeration(RT(S),N(S))\\
&RT(S) = \{RT(S,t_1),RT(S,t_2),\ldots,RT(S,t_K)\}\\
&K = \|LT(S)\|\\
&LT(S) = \{t_1,t_2,\ldots,t_K\}
\end{aligned}
\quad , \qquad (5)
$$

where $N(S)$ - allowed number of passages in section S

$RT(S,t)$ - range of allowed number of passages of type t for section S

$LT(S)$ - sequence of allowed types for section S.

Every passage that appears on a test must have items associated with it appearing. The number of items must be within a specific range.

$$\forall P \in Passages(S) \rightarrow \|Filter(S,\{P\})\| \in RP(S), \qquad (6)$$

where $RP(S)$ is the allowed range of items per passage for section S.

Passages may have "enemy" passages. The enemy relationship is between pairs. Two passages are enemies if they are similar, one provides a clue for the correct answer to the items of other passage, or they should not be in the same section for any other reason.

$$\forall P \in Passages(S) \rightarrow \forall P_i \in Enemies(P) \rightarrow P_i \notin Passages(S), \qquad (7)$$

For each section, the sum of the number of words in passages must be within a specific range.

$$SumWords(Passages(S)) \in RW(S), \qquad (8)$$

where $RW(S)$ is the allowed range of sum of number of words of passages for section S.

Each passage has to be located within a certain range.

$$\forall P_i \in Passages(S) \rightarrow i \in RP(S, Position(P_i)), \qquad (9)$$

where $RP(S,r)$ is the allowed range for position r in section S.

Number of items of each subtype must be within a specific range.

$$\begin{aligned}
&\{\|STFilter(S,st_1)\|,\|STFilter(S,st_2)\|,\ldots,\|STFilter(S,st_K)\|\}\in C\\
&C = Enumeration(RST(S), M(S))\\
&RST(S) = \{RST(S,st_1), RST(S,st_2),\ldots, RST(S,st_K)\}\\
&K = \|LST(S)\|\\
&LST(S) = \{st_1, st_2,\ldots, st_K\}
\end{aligned} \qquad (10)$$

where

$M(S)$ - allowed number of items in section S

$RST(S,st)$ - range of allowed number of items of subtype st for section S

$LST(S)$ - sequence of allowed subtypes for section S

Items may have "enemy" items. The enemy relationship is between pairs. Two items are enemies if they are similar, one provides a clue for the correct answer to the other, or they should not be in the same section for any other reason.

$$\forall I \in S \rightarrow \forall I_i \in Enemies(I) \rightarrow I_i \notin S \qquad (11)$$

For each section, the sum of responses of items must be within a certain range.

$$\forall \theta \in [\theta_L, \theta_U] \rightarrow \sum_{I \in S} Response(I,\theta) \in RResponse(S,\theta), \qquad (12)$$

where θ is ability from $[\theta_L, \theta_U]$ sampled with $\Delta\theta$. The function $Response(I,\theta)$ calculates item response for a given ability, a 3-parameter IRT model is used. The function $RResponse(S,\theta)$ gives the allowed range of summarized responses for a given ability for a given section.

For each section, the sum of information of items must be within a certain range.

$$\forall \theta \in [\theta_L, \theta_U] \rightarrow \sum_{I \in S} Information(I,\theta) \in RInformation(S,\theta), \qquad (13)$$

where θ is ability from $[\theta_L, \theta_U]$ sampled with $\Delta\theta$. The function $Information(I,\theta)$ calculates item information for a given ability. The function $RInformation(S,\theta)$ gives the allowed range of summarized information for a given ability for a given section.

[0034] Constraint Checking Functions

[0035] To facilitate description of the algorithm for some of the constraints, we introduce the following corresponding functions:

Checking constraint (6)

$CheckNumberOfItems(G,U,S)$ $\{G$ - given sequence of passages, U - given sequence of items, S - given section$\}$

Step 1: return $true$ if $\forall P \in G \rightarrow \|Filter(U,\{P\})\| \in RP(S)$ (this can be done in $O(\|G\|+\|U\|)$ time), otherwise $false$

Checking constraint (7)

$CheckEnemies(P,G)$ $\{P$ -checked passage, G -given sequence of passages$\}$

Step 1: return $true$ if $\forall P_i \in Enemies(P) \rightarrow P_i \notin G$ (this can be done in $O(\|Enemies(P)\|)$ time), otherwise $false$

Checking constraint (9)

$CheckPosition(G,S)$ $\{G$ - given sequence of passages, S - given section$\}$

Step 1: sort G by using $Position(P)$ as a sorting key

Step 2: if $\forall P_i \in G \rightarrow i \in RP(S, Position(P_i))$ then return $true$ else return $false$

Setting an order of items with respect to a given sequence of passages

$SetOrder(U,G)$ $\{U$ - given sequence of items, G - given sequence of passages$\}$

Step 1: sort U with respect to order in G (this can be done in $O(\|U\|+\|G\|)$ time)

Checking constraint (11)

$CheckEnemies(I,U)$ $\{I$ -checked item, U -given sequence of items$\}$

Step 1: return *true* if $\forall I_i \in Enemies(I) \to I_i \notin U$ (this can be done in $O(\|Enemies(I)\|)$ time), otherwise *false*

Checking constraints (12),(13)

$CheckIRT(U,S)$ $\{U$ -given sequence of items, S - given section$\}$

Step 1: For each θ from $[\theta_L, \theta_U]$ sampled with $\Delta\theta$ do

Step 1.2: if $\sum_{I \in U} Responce(I,\theta) \notin RResponce(S,\theta)$ OR $\sum_{I \in U} Information(I,\theta) \notin RInformation(S,\theta)$ then return *false*

Step 2: return *true*

[0036] Random Search Algorithm for Test Assembly (Basic Assembly Method)

[0037] The following describes a method of test assembly through random searching in accordance with one embodiment of the present invention:

Step 1. Initialize uniform distribution generator

Step 2. $t_{\max}$ - maximum number of trials

Step 3. $t = 0$ - current trial

Step 4. Generate random sequence V of items from item pool $\mathfrak{I}$

Step 5. if V satisfies constraints (1-13) then return V

Step 6. $t = t + 1$

Step 7. if $t < t_{\max}$ then go to Step 4

Step 8. return $\varnothing$ {search fails}

[0038] Figure 1 is a flow chart illustrating a method for assembling a sub-pool, which can be a test or a section or any portion thereof, in accordance with the present invention. With reference to Figure 1, in step 101, a candidate sub-pool is formed by randomly selecting a plurality of questions from the pool. In step 102, the candidate sub-pool is tested against the constraints. In step 103, if the constraints are satisfied, the candidate

sub-pool is stored as the sub-pool in step 104. If the constraints are not satisfied, in step 103, the process returns to step 101.

[0039] The main disadvantage of the above method is its extremely slow convergence. For example, if test has $m = 10$ items from pool having size $n = 100$ then number of possible candidate item combinations for a single test is

$C_n^m = \frac{n!}{m!(n-m)!} = 17310309456440$. If one were to assume that each item can be used in $k = 1000000$ different overlapping tests, the probability of hitting a feasible solution is $\frac{nk}{mC_n^m} = 5.7769\text{E}-07$, which is practically 0.

[0040] **Methods to Shrink Search Domain**

[0041] Referring first to Figure 2, two sets A and $B \subset A$ are illustrated. Set A (the search region) consists of all possible combinations of items, satisfying a particular constraint, and its subset B consists of all combinations of items resulting in a test. The pure random search is based on uniform distribution and converges to a test with probability $P = \frac{|B|}{|A|}$. Thus, if set A is shrunk without losing combinations from B, then P will increase and, consequently, the speed of the random search test assembly will increase.

[0042] The following describes three methods that have been developed to shrink set A:

[0043] **Method 1 (Search region decomposition):** A test consists of sections, which may include sequences of passages or may simply include items which can be grouped; each passage, if any, introduces a set of items, which can be grouped. Taking into account this hierarchical structure of a test and the "divide and conquer" principle, the size of the search region can be substantially reduced. In particular, the problem is separated into a sequence of less computationally complex sub-problems such that, for each of them, n and m are relatively small. Thus, whole sets of infeasible item combinations can be rejected. In addition, checking of constraints is prioritized in such a way that the most computationally easy constraints are checked first.

[0044] With reference to Figure 3, set A is reduced into search regions for the sections of each type (in this example, section type are AR, RC and LR). For example, search region AR consists of all combinations of AR items, where each combination must include a bounded number of items: $|A| >> |AR| + |RC| + |LR|$.

[0045] With reference to Figure 4, the search region LR is reduced into search regions for the groups of items corresponding to the constraint (10). Here, sub-region S_i has all combinations of LR items for the subtype i, where each combination includes a bounded number of items: $|LR| >> \sum_{i=1}^{n} |S_i|$.

[0046] With reference to Figure 5A, a tree 530 with multiple nodes is illustrated. The tree 530 may represent a test or any portion of a test, such as a section. The tree 530 includes terminal nodes 531, a root node 532 and may include other intermediate nodes 533, 534. Each node is associated with constraints for a particular part of a test or portion thereof, represented by the tree 530. Thus, where the tree 530 represents the whole test, the root node 532 has constraints for the whole test (e.g., the allowed number of items in the test and/or score). Intermediate node 533, representing a section in the tree illustrated, is associated with constraints for the section. Similarly, intermediate node 534, representing a passage in the tree illustrated, is associated with constraints for the passage. Terminal nodes 531 similarly are associated with constraints.

[0047] As discussed below with reference to Figures 5B and 5C, the assembly process starts from the terminal nodes 531 (each a "leaf") and goes up to the root node 532. Items corresponding to each leaf are selected. In the non-terminal nodes (533, 534), sub-parts from sons are concatenated and then checked against the node's constraints. If the constraints are satisfied, the concatenation result is sent up to the node's father (going up); sons violating the constraints are forced to reassemble their parts (going down). The process is repeated up and down along the tree 530 until the constraints in the root node 532 are satisfied.

[0048] Further shown is two scenarios (Figures 5B, 5C) of assembling a sub-pool from a tree or its arbitrary sub-tree. With reference to Figure 5B, in step 501, a hierarchical representation of the sub-pool is created. As illustrated in Figure 5A, the hierarchical representation comprises a root node and at least one other node. At least one of the

other nodes comprises a terminal node. The root node is associated with one or more root constraints and each of the other nodes are associated with one or more additional constraints. In step 502, a candidate question set is formed, which includes randomly selecting a plurality of questions from the pool. In step 503, the candidate question set is tested against the additional constraints associated with the terminal node. If the additional constraints are satisfied (step 504), in step 505, at least the candidate question set is tested against the root constraints. To the extent there exists more than one terminal node, steps 502, 503 and 504 are repeated with respect to such terminal nodes.

[0049] With reference to Figure 5C, a hierarchical representation of the sub-pool is created in step 506. The hierarchical representation comprises a root node and at least two other nodes. At least two of the other nodes each comprise a terminal node. The root node is associated with one or more root constraints and each of the other nodes are associated with one or more additional constraints. In step 507, a first candidate question set is formed, which includes randomly selecting a plurality of questions from the pool. In step 508, the first candidate question set is tested against the additional constraints associated with a first of the terminal nodes. In step 509, a second candidate question set is formed, which includes randomly selecting a plurality of questions from the pool. In step 510, the second candidate question set is tested against the additional constraints associated with a second of the terminal nodes. If the first set of additional constraints and the second set of additional constraints are satisfied (step 511), the first candidate question set and the second candidate question set are concatenated in step 512 to form a combined question set. In step 513, at least the combined set is tested against the root constraints. If the first set of additional constraints is not satisfied in step 511, steps 507 and 508 are repeated. If in step 511 the second set of additional constraints is not satisfied, steps 509 and 510 are repeated.

[0050] **Method 2 (Constraints enumeration):** Constraints (1), (2), (5), (10) can be automatically satisfied by using function $Enumeration(R,s)$ (see its definition above). Consider an example of constraints (1) and (2):

$$\begin{array}{l} M = 2 \\ N = 10 \\ RI(S_1) = [4,8] \\ RI(S_2) = [3,7] \end{array} \text{ then } C = Enumeration(\{RI(S_1), RI(S_2)\}, N) = (\{4,6\},\{5,5\},\{6,4\},\{7,3\}).$$

[0051] Randomly selected element $\{i_1, i_2\}$ from C provides satisfaction for constraints (1), (2) and reduces search region to a set of tests consisting of two sections with i_1 and i_2 number of items, respectively.

[0052] Figure 6 illustrates a method involving constraints enumeration. In step 601, a sequence of ranges is formed, wherein each range in the sequence imposes a constraint on a scalar property of the sub-pool. In step 602, a vector comprising a plurality of elements is randomly formed, wherein each of the elements in the vector belongs to at least one range in the sequence. In step 603, a plurality of questions are randomly selected from the pool to form the sub-pool such that each of the scalar properties of the sub-pool is equal to at least one of the elements of the vector.

[0053] **Method 3 (Search region greedy reduction):** For computationally easy constraints (1-11), a simple greedy heuristics based on tabu search can be used. If a random combination of passages/items does not satisfy (1-11), this combination is removed from the passage/item pool to passage/item tabu region. After the combination obeying (1-11) is found, or the pools are exhausted, the passage/item tabu region is removed back to the passage/item pool (as illustrated in Figures 7, 8).

[0054] Figure 8 illustrates a method involving search region greedy reduction. In step 801, a candidate sub-pool is formed by randomly selecting a plurality of questions from the pool. In step 802, it is determined if the candidate sub-pool satisfies the constraints. If the constraints are not satisfied, in step 803 the questions of the candidate sub-pool are removed from the pool of questions and steps 801 and 802 are repeated. If the constraints are satisfied, the candidate sub-pool is stored as the sub-pool in step 804. When all of the questions from the pool of questions have been removed, the pool is restored with all of the removed questions in step 805 and the process repeats.

[0055] The one or more of the methods illustrated in Figures 1, 5A, 5B, 5C, 6 and 8 may be combined in a single process in accordance with the present invention.

[0056] Test Assembly Algorithm (TA – Algorithm)

[0057] The following describes a test assembly algorithm in accordance with one embodiment of the present invention:

$AssembleTest(N, M, R)$ {assembles and returns test T, N - given number of items per test, M - given number of sections per test, $R = \{r_1, r_2, ..., r_M\}$ - given sequence of allowed ranges of number of items per section}

Step 1: $C = Enumeration(R, N)$ {vector of sequences of allowed number of items per section, provides (1), (2)}

Step 2: $c = Random(C)$

Step 3: Initialize T

Step 4: for each $n \in c$ do

Step 4.1: $S = AssembleSection(n)$ {see Section Assembly Algorithm}

Step 4.2: if $S \neq \varnothing$ then add S to T *else go to Step 4.1*

Step 5: if $Score(T) \in RS$ {provides (4)} then return T else return items and passages from T back to the pools

Step 6: go to Step 2

[0058] Section Assembly Algorithm

[0059] The following describes a section assembly algorithm in accordance with one embodiment of the present invention:

$AssembleSection(n)$ {assembles and returns section S, n - given number of items in section}

Step 1: Initialize S

Step 2: $C = Enumeration(RT(S), N(S))$ {vector of sequences of allowed number of passages per type, provides (5)}

Step 3: $c = Random(C)$

Step 4: $G = AssemblePassages(c, S)$ {see Sequence of Passages Assembly Algorithm}

Step 5: if $G = \varnothing$ then go to Step 3

Step 6: $U = AssembleItems(G, S, n)$ {see Sequence of Items Assembly Algorithm}

Step 7: if $U = \varnothing$ then go to Step 3

Step 8: if $CheckIRT(U, S)$ {provides (12), (13)} then

Step 8.1: $\mathfrak{R} = \mathfrak{R} \setminus G, \mathfrak{I} = \mathfrak{I} \setminus U$ {provides (3)}

Step 8.2: Add U to S

Step 8.3: return S

Step 9: go to Step 3

[0060] Sequence of Passages Assembly Algorithm

[0061] The following describes an algorithm for assembling a sequence of passages in accordance with one embodiment of the present invention:

$AssemblePassages(c, S)$ {assembles and returns sequence of passages G, c - given sequence of allowed number of passages per type, S - given section}

Step 1: $\aleph = \varnothing$ {setup tabu region}

Step 2: $G = \varnothing$

Step 3: for each $n \in c, t \in LT(S)$ do { n is current allowed number of passages having type t }

Step 3.1: $Q = TFilter(\mathfrak{R}, t)$ {subsequence of all available passages having type t}

Step 3.2: for $j = 1$ to n do

Step 3.2.1: $P = Random(Q)$

Step 3.2.2: if $CheckEnemies(P, G)$ {provides (7)} then go to Step 3.2.3 *else go to Step 3.2.1*

Step 3.2.3: $Q = Q \setminus \{P\}$ {provides (3)}

Step 3.2.4: $G = G \cup \{P\}$

Step 3.2.5: $\Re = \Re \setminus \{P\}, \aleph = \aleph \cup \{P\}$ {expand tabu region}

Step 4: if $SumWords(G) \in RW(S)$ {provides (8)} AND $CheckPosition(G,S)$ {provides (9)} then

Step 4.1: $\Re = \Re \cup \aleph$ {remove passages from tabu region}

Step 4.2: return G

Step 5: go to Step 2

[0062] Sequence of Items Assembly Algorithm

[0063] The following describes an algorithm for assembling a sequence of items in accordance with one embodiment of the present invention:

$AssembleItems(G,S,n)$ {assembles and returns sequence of items U, G - given sequence of passages, S - given section, n - given number of items}

Step 1: $C = Enumeration(RST(S),n)$ {vector of sequences of allowed number of items per subtype, provides (10)}

Step 2: $Q = Filtr(\Im, G)$ {subsequence of items related to passages from G }

Step 3: $U = \varnothing$

Step 4: $c = Random(C)$

Step 5: for each $m \in c, st \in LST(S)$ do { m is current allowed number of items having subtype st }

Step 5.1: $W = STFilter(Q, st)$ {subsequence of items having subtype st }

Step 5.2: for $j = 1$ to m do

Step 5.2.1: $I = Random(W)$

Step 5.2.2: if $CheckEnemies(I,U)$ {provides (11)} then go to Step 5.2.3 *else go to Step 5.2.1*

Step 5.2.3: $W = W \setminus \{I\}$ {provides (3)}

Step 5.2.4: $U = U \cup \{I\}$

Step 5.2.5: $Q = Q \setminus \{I\}$ {expand tabu region}

Step 6: if $CheckNumberOfItems(G, U, S)$ then

Step 6.1: $SetOrder(U, G)$

Step 6.2: return U

Step 7: go to Step 3

[0064] The test resulting from implementation of the above algorithms satisfies all constraints (1-13).

[0065] Special Section Case

[0066] In one special situation, a section has passages of one type and each passage relates to exactly one item and vice versa. In this case, constraints (5-7) can be omitted. $AssembleSection(n)$ {assembles and returns section S, n - given number of items in the section}

Step 1: Initialize S

Step 2: $C = Enumeration(RST(S), n)$ {vector of sequences of allowed number of items per subtype, provides (10)}

Step 3: $\aleph = \varnothing$ {setup tabu region}

Step 4: $U = \varnothing$

Step 5: $c = Random(C)$

Step 6: for each $m \in c, st \in LST(S)$ do { m is current allowed number of items having subtype st }

Step 6.1: $W = STFilter(\Im, st)$ {subsequence of available items having subtype st }

Step 6.2: for $j = 1$ to m do

Step 6.2.1: $I = Random(W)$

Step 6.2.2: if $CheckEnemies(I, U)$ {provides (11)} then go to Step 6.2.3 *else go to Step 6.2.1*

Step 6.2.3: $W = W \setminus \{I\}$ {provides (3)}

Step 6.2.4: $U = U \cup \{I\}$

Step 6.2.5: $\Im = \Im \setminus \{I\}$, $\aleph = \aleph \cup \{I\}$ {expand tabu region}

Step 7: $G = Passages(U)$ {this can be done in $O(\|U\|)$ time}

Step 8: if $SumWords(G) \in RW(S)$ {provides (8)} AND $CheckPosition(G,S)$ {provides (9)} then

Step 8.1: $\Im = \Im \cup \aleph$ {return items from tabu region}

Step 8.2: if $CheckIRT(U,S)$ {provides (12), (13)} then

Step 8.2.1: $\Im = \Im \setminus U$ {provides (3)}

Step 8.2.2: Add U to S

Step 8.2.3: return S

Step 8.3: go to Step 3

Step 9: go to Step 4

[0067] Implementation Issues

[0068] In all infinite loops (see above), there should be counters limiting the number of possible iterations, over-iteration and search failure handlers. These are avoided to make all algorithms more clear. The C/C++ implementation has items and passages located in $\Im$ and $\Re$ pools. The rest of the sets are sequences of corresponding indices. Also, most of the set operations are avoided by using Boolean properties for each item and passage. For example, when an item is assigned to the test its property "Taken" is set to true.

[0069] Analysis of TA Algorithm

[0070] The convergence of the algorithm fully depends on the persistence of the infinite loops and the size of the domain (set B in Figure 2) of item combinations satisfying constraints (1-13).

[0071] The algorithm can cope with non-linear constraints, which create an intractable problem for the MIP approach. The time to find a solution with the random search

algorithm will increase because nonlinear functions must be evaluated, but the fundamental approach remains unchanged.

[0072] Item Pool Analysis Algorithm

[0073] The TA - Algorithm (described above) can be used to sequentially generate non-overlapping tests. Using this method, items and passages used in the current test are removed from the respective pools $\mathfrak{I}$ and $\mathfrak{R}$ before assembling the next test. An example is discussed with reference to Figure 9. In this example, each section includes two passages; Figure 9 shows all possible sections as segments that can overlap each other (i.e., have common passages), such as, for example, sections (1, 3) and (1, 4). Using the test assembly algorithm sequentially, the sections that will be generated next cannot be controlled. In the worst case, section 1 is assembled first. The next section to be assembled can only be section 2, because passages of sections 3, 4 and 5 are already used by section 1. Thus, this approach yields only two sections, although there are three possible non-overlapping sections (i.e., 2, 3, 4 or 2, 3, 5).

[0074] This example leads to the inventive approach for solving the extraction problem for sections, which includes two phases. In the first phase, the set of all possible sections is generated. In the second phase, a maximum subset of non-overlapping sections is identified. The second phase can be formulated and solved as maximum clique or maximum set packing problem. This approach provides more non-overlapping sections than the sequential approach described above. Generating all possible non-overlapping sections is not practically possible for the large pools; however, by using the Monte-Carlo technique, described herein, a representative set can be built, which will provide a strong lower bound.

[0075] With reference to Figure 10A a method of assembling a plurality of disjoint sub-pools from a pool of questions is illustrated. Each sub-pool includes a plurality of questions and satisfies one or more constraints. In step 1001, a first collection of intersecting sub-pools is assembled. This assembly can be accomplished by the inventive random search methods described herein (see Figure 10B), or by some prior art method, within the scope of the present invention. In step 1002, a second collection of sub-pools is extracted from the first collection of sub-pools. The second collection of sub-pools comprises the plurality of mutually disjoint sub-pools. In step 1003, the second

collection of sub-pools is stored. In some embodiments, with reference to Figure 10B, step 1001 comprises: step 1004, forming a candidate sub-pool by randomly selecting a plurality of questions from the pool; step 1005, testing the candidate sub-pool against the constraints; and step 1006, if the constraints are satisfied, storing (in step 1007) the candidate sub-pool as the sub-pool.

[0076] Referring back to Figure 9 and considering the usage frequency F of each passage, it is determined that the most frequently used passages are c ($F=3$), then b ($F=2$), then a,d,e,f,g ($F=1$) and h,i,j,k,l ($F=0$). If assembling is started from the passages having a frequency $F \le 1$ ($\Im=\{a,d,e,f,g,h,i,j,k,l\}$), section 2 is obtained. Then, passages with frequency $F=2$ ($\Im=\{a,d,g,h,i,j,k,l,b\}$) are added to obtain section 3. Finally, passages with frequency $F=3$ ($\Im=\{d,g,h,i,j,k,l,c\}$) are added to obtain section 4 or 5. Three sections are assembled, which is maximum number of non-overlapping sections; the optimal solution for the extraction problem is obtained.

[0077] The following provides algorithms for solving the extraction problem for tests and sections in accordance with one embodiment of the present invention.

[0078] Algorithm for Tests Extraction

Step 1: Solve extraction problem for sections of each type (see Algorithms for Extraction of Sections below)

Step 2: Assemble set $\aleph$ of overlapping tests based on extracted sections and taking into account expected test score constraint (4) and the constraint on number of the items in the test

Step 3: Solve maximum clique problem for $\aleph$

[0079] Algorithm for Extraction of Sections Comprising Set-Based Items

[0080] The following algorithm is for extraction of the sections comprising set-based items:

Step 1: Assemble representative set $\aleph$ of all overlapping sections

Step 2: Solve maximum clique problem for $\aleph$

[0081] Algorithm for Extraction of Sections Comprising Discrete Items

[0082] The following algorithm is for extraction of the sections comprising discrete items:

Step 1: Assemble sequentially large set of overlapping sections calculating usage frequency for each passage (step 8.1 in the Section Assembly Algorithm is avoided)

Step 2: Find maximum frequency $F_{\max}$

Step 3: Extract non-overlapping sections sequentially based on the usage frequency:

Step 3.1: $\tilde{\mathfrak{R}} = \varnothing$

Step 3.2: for $j = 0$ to $F_{\max}$ do

Step 3.2.1: $\tilde{\mathfrak{R}} = \tilde{\mathfrak{R}} \cup \{$passages from $\mathfrak{R}$ having frequency $j\}$

Step 3.2.2: Assemble non-overlapping sections sequentially for current $\tilde{\mathfrak{R}}$, $\mathfrak{I}$ and the limited number of trials

[0083] For the special case of sections (as described above in Special Section Case), the items should be operated with directly, i.e., $\tilde{\mathfrak{R}}$ should be $\tilde{\mathfrak{I}}$.

[0084] Assessing Strength of Item Pools / Assessing Future Needs

[0085] Solving the problem of identifying multiple non-overlapping tests provides a robust measure of the item bank and can guide in the development of new items. In particular, all tests available from the pool could be assembled and those items not chosen in the assembly could be analyzed. The analysis is based on running automatic test assembly software and identifying the most frequently violated section/test constraints. Those constraints infer desired properties for future items to be developed for the pool. Items are a valuable commodity and the future development of items should not duplicate the attributes of the unused items. TA-Algorithm can assemble multiple overlapping tests. A collateral result of this process is a determination of the number of times each item was used (i.e., usage frequency), which corresponds to weakly (high frequency) or strongly (low frequency) represented clusters in item parametric space. Combined with an analysis of violations of the constraints, it provides valuable information about the kind of items needed in the pool to assemble more tests.

[0086] As illustrated in Figure 10, the method of assembling a plurality of disjoint sub-pools also allows for analysis of the item pool. In particular, in step 1008, for each of the questions in the first collection of sub-pools, a frequency of usage in the first collection of sub-pools is computed. In step 1009, the questions in the pool of questions are analyzed based on the computed frequency.

[0087] It should be understood that various alternatives and modifications of the present invention could be devised by those skilled in the art. Nevertheless, the present invention is intended to embrace all such alternatives, modifications and variances that fall within the scope of the appended claims.